```
  1: { ****************************************************** }
  2: {                                                        }
  3: { Delphi/NET Runtime Library                             }
  4: {                                                        }
  5: {             }
  6: {                                                        }
  7: { All Rights Reserved.                                   }
  8: {                                                        }
  9: { ****************************************************** }
 10:
 11:
 12: unit Borland.Vcl.Design.Proxies;
 13:
 14: interface
 15:
 16: uses
 17:    System.Collections, System.Reflection, System.Reflection.Emit,
 18:    System.Globalization, TypInfo, Classes, SysUtils;
 19:
 20:
 21: //!! APIs have changed quite a bit
 22: function CreateSubClass(AAncestor: TClass; const AClassName: strin
g;
 23:    const AUnitName: string = ''): TClass;
 24: procedure DestroySubClass(AInstance: TObject); overload; deprecate
d;
 25: procedure DestroySubClass(AClass: TClass); overload;
 26: procedure RenameSubClass(AInstance: TObject; const AClassName: str
ing;
 27:    const AUnitName: string = ''); overload; deprecated;
 28: procedure RenameSubClass(AClass: TClass; const AClassName: string;
 29:    const AUnitName: string = ''); overload;
 30:
 31: // TODO: ConstructSubClass - this should not be needed!
 32: function ConstructSubClass(AClass: TClass; AParams: array of
TObject): TObject;
 33: // TODO: ConstructComponent - this should not be needed!
 34: function ConstructComponent(AClass: TComponentClass; AOwner:
TComponent = nil): TComponent;
 35:
 36: function IsProxyClass(AInstance: TObject): Boolean; overload;
 37: function IsProxyClass(AClass: TClass): Boolean; overload;
 38:
 39: // TODO: ChangeToProxyClass, this can't work like the old way so w
ill
this do?
 40: procedure ChangeToProxyClass(AInstance: TObject{; TClass argument}
);
overload; deprecated;
 41: procedure ChangeToProxyClass(AClass: TClass); overload;
 42:
 43: function CreateSubClassMethod(AInstance: TObject;
 44:    const AMethodName: string): TMethodCode;
 45: procedure RenameSubClassMethod(AInstance: TObject;
 46:    const AMethodCode: TMethodCode; const AMethodName: string);
 47: procedure DestroySubClassMethod(AInstance: TObject;
 48:    const AMethodCode: TMethodCode);
 49:
 50: procedure HandleNotification(Sender: TObject; AComponent: TCompone
nt;
Operation: TOperation);
 51:
 52: procedure SaveIt;
 53:
 54: type
```

```
 55:    EProxyError = class(Exception);
 56:
 57: implementation
 58:
 59: uses System.Runtime.InteropServices;
 60:
 61: type
 62:    TProxyIntercept = class(TObject, IProxySystemSupport,
IProxyTypInfoSupport)
 63:    strict private
 64:       function GetMethodAddress(AClass: TClass; const AName: string;
out ACode: TMethodCode): Boolean;
 65:
 66:       function GetMethodProp(AInstance: TObject; APropInfo: TPropInf
o;
out AMethod: TMethod): Boolean;
 67:       function SetMethodProp(AInstance: TObject; APropInfo: TPropInf
o;
const AMethod: TMethod): Boolean;
 68:       function GetUnitName(ATypeInfo: TTypeInfo; out AUnitName:
string): Boolean;
 69:    end;
 70:
 71:    TInstanceRef = class(TObject)
 72:    public
 73:       Props: Hashtable;
 74:       constructor Create;
 75:    end;
 76:
 77:    TProxyType = class(TypeDelegator)
 78:    strict private
 79:       class var
 80:          FAssemblyBuilder: AssemblyBuilder;
 81:          FModuleBuilder: ModuleBuilder;
 82:          FProxyTypeIndex: Integer;
 83:          FProxyIntercept: TProxyIntercept;
 84:          FRootMetaType: System.Type;
 85:          FRootHandleField: FieldInfo;
 86:          FRootParentField: FieldInfo;
 87:          FProxyNotificationMethod: MethodInfo;
 88:          FSendNotificationMethod: MethodInfo;
 89:          FProxies: Hashtable;
 90:          FInstances: Hashtable;
 91:
 92:       var
 93:          FClassName: string;
 94:          FUnitName: string;
 95:          FMethods: Hashtable;
 96:
 97:    strict protected
 98:       class procedure CreateBoolAttribute(ATypeBuilder: TypeBuilder;
 99:          AAttribute: System.Type; AValue: Boolean = True);
100:       class function CreateMetaSubType(ABaseType, AType: System.Type
;
101:          ATypeBuilder: TypeBuilder): System.Type;
102:       class procedure CodeGenConstructors(ABaseType: System.Type;
ATypeBuilder: TypeBuilder);
103:       class procedure CodeGenNotification(ABaseType: System.Type;
ATypeBuilder: TypeBuilder);
104:       class function FindRealType(var AType: System.Type): Boolean;
105:    public
106:       class constructor Create;
107:       constructor Create(Ancestor: System.Type; const AClassName,
AUnitName: string);
108:
109:       // delegator work
110:       function get_FullName: string; override;
```

```
111:        function get_Name: string; override;
112:        function get_Namespace: string; override;
113:
114:        // support for the public functions
115:        class function FindProxy(AInstance: TObject): TProxyType;
116:        function CreateMethod(const AMethodName: string): TMethodCode;
117:        procedure RenameMethod(const AMethodCode: TMethodCode; const
AMethodName: string);
118:        procedure DestroyMethod(const AMethodCode: TMethodCode);
119:
120:        // type versions of the public functions
121:        class function IsSubTyped(AType: System.Type): Boolean;
122:        class function CreateSubType(ABaseType: System.Type; const
AClassName: string;
123:          const AUnitName: string = ''): System.Type;
124:        class procedure ChangeToProxyType(AType: System.Type);
125:        class procedure DestroySubType(AType: System.Type);
126:        class procedure RenameSubType(AType: System.Type; const
AClassName: string;
127:          const AUnitName: string = '');
128:
129:        // support functions for TProxyIntercept
130:        class function GetMethodAddress(AClass: TClass; const AName:
string; out ACode: TMethodCode): Boolean;
131:        class function GetMethodProp(AInstance: TObject; APropInfo:
TPropInfo; out AMethod: TMethod): Boolean;
132:        class function SetMethodProp(AInstance: TObject; APropInfo:
TPropInfo; const AMethod: TMethod): Boolean;
133:        class function GetUnitName(ATypeInfo: TTypeInfo; out AUnitName
:
string): Boolean;
134:
135:        class procedure HandleNotification(Sender: TObject; AComponent
:
TComponent; Operation: TOperation); static;
136:
137:        // onetime snapshot of Proxies' scratch assembly
138:        // WARNING: once you 'SaveIt'; you can't create anymore proxy
classes/types
139:        class procedure SaveIt;
140:      end;
141:
142:      TObjects = array of TObject;
143:      TMethodProxy = class(TMethodCode)
144:      strict private
145:        FProxyType: TProxyType;
146:        FName: string;
147:      public
148:        constructor Create(AProxyType: TProxyType; const AName: string
);
149:        procedure Clear;
150:
151:        // TMethodProxy stuff
152:        procedure Rename(Value: string);
153:        function get_ProxyType: TProxyType;
154:        property ProxyType: TProxyType read get_ProxyType;
155:
156:        // MemberInfo stuff
157:        function GetCustomAttributes(AInherit: Boolean): TObjects;
override;
158:        function GetCustomAttributes(AttributeType: System.Type; Inher
it:
Boolean): TObjects; override;
159:        function IsDefined(AttributeType: System.Type; Inherit: Boolea
n):
Boolean; override;
160:        function get_DeclaringType: System.Type; override;
161:        function get_MemberType: MemberTypes; override;
```

```
162:        function get_Name: string; override;
163:        function get_ReflectedType: System.Type; override;
164:        property DeclaringType: System.Type read get_DeclaringType;
165:        property MemberType: MemberTypes read get_MemberType;
166:        property Name: string read get_Name;
167:        property ReflectedType: System.Type read get_ReflectedType;
168:    end;
169:
170: { TProxyIntercept }
171:
172: function TProxyIntercept.GetMethodAddress(AClass: TClass; const
AName: string; out ACode: TMethodCode): Boolean;
173: begin
174:     Result := TProxyType.GetMethodAddress(AClass, AName, ACode);
175: end;
176:
177: function TProxyIntercept.GetMethodProp(AInstance: TObject; APropIn
fo:
TPropInfo; out AMethod: TMethod): Boolean;
178: begin
179:     Result := TProxyType.GetMethodProp(AInstance, APropInfo, AMethod
);
180: end;
181:
182: function TProxyIntercept.SetMethodProp(AInstance: TObject; APropIn
fo:
TPropInfo; const AMethod: TMethod): Boolean;
183: begin
184:     Result := TProxyType.SetMethodProp(AInstance, APropInfo, AMethod
);
185: end;
186:
187: function TProxyIntercept.GetUnitName(ATypeInfo: TTypeInfo; out
AUnitName: string): Boolean;
188: begin
189:     Result := TProxyType.GetUnitName(ATypeInfo, AUnitName);
190: end;
191:
192: { TInstanceRef }
193:
194: constructor TInstanceRef.Create;
195: begin
196:     inherited;
197:     Props := Hashtable.Create;
198: end;
199:
200:
201: { TProxyType }
202:
203: const
204:     STestAssemblyName = 'VclDesignTime_ProxyAssembly';
205:     STestModuleName = 'VclDesignTime_ProxyModule';
206:     STestTypeName = 'Borland.Vcl.DesignTime.ProxyType%d';
207:     STestFileName = STestAssemblyName + '.dll';
208:
209: var
210:     EchoLevel: Integer = 0;
211:
212: {procedure EchoType(const APrefix: string; AType: System.Type;
AMaxDepth: Integer = 4);
213: begin
214:     Inc(EchoLevel);
215:     try
216:         WriteLn(APrefix, ' ****************');
217:         if EchoLevel > AMaxDepth then
218:             WriteLn(APrefix, ' IS TOO DEEP')
219:         else
220:             if AType = nil then
```

```
221:            WriteLn(APrefix, ' IS NIL')
222:          else
223:          begin
224:            WriteLn(APrefix, '.Name = ', AType.Name);
225:            WriteLn(APrefix, '.FullName = ', AType.FullName);
226:            WriteLn(APrefix, '.Assembly = ', AType.Assembly.FullName);
227:            WriteLn(APrefix, '.AssemblyQualifedName = ',
AType.AssemblyQualifedName);
228:            WriteLn(APrefix, '.NameSpace = ', AType.NameSpace);
229:            WriteLn(APrefix, '.Attributes = ',
System.Enum(AType.Attributes).ToString);
230:            WriteLn(APrefix, '.MemberType = ',
System.Enum(AType.MemberType).ToString);
231:            try
232:              WriteLn(APrefix, '.TypeHandle = ', AType.TypeHandle.Valu
e);
233:            except
234:              on E: Exception do
235:                WriteLn(APrefix, '.TypeHandle = ', E.Message);
236:            end;
237:            try
238:              WriteLn(APrefix, '.ClassName = ', AType.ClassName);
239:              WriteLn(APrefix, '.ClassInfo.Name = ',
AType.ClassInfo.Name);
240:            except
241:              on E: Exception do
242:                begin
243:                  WriteLn(APrefix, '.ClassName = ', E.Message);
244:                  WriteLn(APrefix, '.ClassInfo.Name = ', E.Message);
245:                end;
246:            end;
247:            if AType.Module <> nil then
248:              WriteLn(APrefix, '.Module.Name = ', AType.Module.Name);
249:            if AType.BaseType <> nil then
250:              EchoType(APrefix + '.BaseType', AType.BaseType, AMaxDept
h);
251:            if AType.DeclaringType <> nil then
252:              EchoType(APrefix + '.DeclaringType', AType.DeclaringType
,
AMaxDepth);
253:            if AType.UnderlyingSystemType <> nil then
254:              EchoType(APrefix + '.UnderlyingSystemType',
AType.UnderlyingSystemType, AMaxDepth);
255:          end;
256:      finally
257:        Dec(EchoLevel);
258:        if EchoLevel < 0 then
259:        begin
260:          WriteLn('##################### How did that happen?
#####################');
261:          EchoLevel := 0;
262:        end;
263:      end;
264: end;}
265:
266: resourcestring
267:     SNoHandleNotification = 'Could not find
Borland.Vcl.Design.Proxies.Unit.HandleNotification';
268:     SNoSendNotification = 'Could not find
Borland.Vcl.Classes.Unit.SendNotification';
269:     SCouldNotFindBaseMeta = 'Could not find BaseMetaClass';
270:     SCouldNotFindTypeHandle = 'Could not find
RootMetaClass.FInstanceTypeHandle';
271:     SCouldNotFindParent = 'Could not find RootMetaClass.FClassParent
';
272:     SCouldNotFindConstructor = 'Could not find BaseType.Constructor'
;
273:     SCouldNotFindMetaConstructor = 'Could not find
```

```
MetaClass.Constructor';
 274:    SAlreadyProxy = 'Type is already a proxy';
 275:    STypeNotSubType = 'Type is not a subtype';
 276:    SMethodNotMethodProxy = 'Method is not a method proxy';
 277:
 278:
 279: class constructor TProxyType.Create;
 280: var
 281:    LAssemblyName: AssemblyName;
 282:    LProxiesUnitType: System.Type;
 283:    LClassesUnitType: System.Type;
 284: begin
 285:    // a place to work
 286:    FProxies := Hashtable.Create;
 287:    FInstances := Hashtable.Create;
 288:
 289:    // create our scratcharea assembly and module
 290:    LAssemblyName := AssemblyName.Create;
 291:    LAssemblyName.Name := STestAssemblyName;
 292:    FAssemblyBuilder :=
AppDomain.CurrentDomain.DefineDynamicAssembly(LAssemblyName,
AssemblyBuilderAccess.RunAndSave);
 293:    FModuleBuilder :=
FAssemblyBuilder.DefineDynamicModule(STestModuleName, STestFileName,
True);
 294:
 295:    // the following is need simply to keep the compiler from
smartlinking certain functions into oblivion
 296:    if FProxyTypeIndex < 0 then
 297:    begin
 298:      Borland.Vcl.Design.Proxies.HandleNotification(nil, nil, opInse
rt);
 299:      Classes.SendNotification(nil, nil, opInsert);
 300:    end;
 301:
 302:    // find the sendnotification function over in Classes
 303:    LProxiesUnitType :=
TypeOf(EProxyError).Assembly.GetType('Borland.Vcl.Design.Proxies.
Unit');
 304:    FProxyNotificationMethod :=
LProxiesUnitType.GetMethod('HandleNotification',
 305:      BindingFlags.Public or BindingFlags.Static or
BindingFlags.InvokeMethod);
 306:    if FProxyNotificationMethod = nil then
 307:      raise EProxyError.Create(SNoHandleNotification);
 308:
 309:    // find the sendnotification function over in Classes
 310:    LClassesUnitType :=
TypeOf(Classes.TOperation).Assembly.GetType('Borland.Vcl.Classes.
Unit');
 311:    FSendNotificationMethod :=
LClassesUnitType.GetMethod('SendNotification',
 312:      BindingFlags.Public or BindingFlags.Static or
BindingFlags.InvokeMethod);
 313:    if FSendNotificationMethod = nil then
 314:      raise EProxyError.Create(SNoSendNotification);
 315:
 316:    // wedge into System and TypInfo
 317:    FProxyIntercept := TProxyIntercept.Create;
 318:    Borland.Delphi.System.ProxySystemSupport := FProxyIntercept;
 319:    ProxyTypInfoSupport := FProxyIntercept;
 320: end;
 321:
 322: constructor TProxyType.Create(Ancestor: System.Type; const AClassN
ame,
 323:    AUnitName: string);
 324: begin
 325:    inherited Create(Ancestor);
```

```
326:    FClassName := AClassName;
327:    FUnitName := AUnitName;
328:    FMethods := Hashtable.Create;
329: end;
330:
331: function TProxyType.get_Name: string;
332: begin
333:    Result := FClassName;
334: end;
335:
336: function TProxyType.get_FullName: string;
337: begin
338:    Result := FUnitName + '.' + FClassName;
339: end;
340:
341: function TProxyType.get_Namespace: string;
342: begin
343:    Result := FUnitName;
344: end;
345:
346: class function TProxyType.IsSubTyped(AType: System.Type): Boolean;
347: begin
348:    // while FindRealType will change the AType we passed it the
349:    //   callee won't see it
350:    Result := FindRealType(AType);
351: end;
352:
353: class procedure TProxyType.CreateBoolAttribute(ATypeBuilder:
TypeBuilder;
354:    AAttribute: System.Type; AValue: Boolean);
355: var
356:    LAttributeConstructor: ConstructorInfo;
357: begin
358:    LAttributeConstructor :=
AAttribute.GetConstructor([TypeOf(AValue)]);
359:
ATypeBuilder.SetCustomAttribute(CustomAttributeBuilder.Create(
LAttributeConstructor, [AValue]));
360: end;
361:
362: class function TProxyType.CreateMetaSubType(ABaseType, AType:
System.Type; ATypeBuilder: TypeBuilder): System.Type;
363: var
364:    LBaseType: System.Type;
365:    LTypeBuilder: TypeBuilder;
366:    LBaseConstructor: ConstructorInfo;
367:    LRootMetaType: System.Type;
368:    LRootHandleField: FieldInfo;
369:    LRootParentField: FieldInfo;
370:    LConstructorBuilder: ConstructorBuilder;
371:    LILGenerator: ILGenerator;
372:    LBaseInstanceField: FieldInfo;
373:    LInstanceField: FieldInfo;
374:    LTypeConstructorBuilder: ConstructorBuilder;
375:
376: begin
377:    // find the base metatypes
378:    LBaseType := ABaseType.GetNestedType('@Meta' + ABaseType.Name);
379:    if LBaseType = nil then
380:      raise EProxyError.Create(SCouldNotFindBaseMeta);
381:
382:    // found the root metatype yet?
383:    if FRootMetaType = nil then
384:    begin
385:
386:      // chase up the metaclass parentage to find the root
387:      FRootMetaType := LBaseType;
388:      while FRootMetaType.BaseType <> nil do
```

```
389:     begin
390:       if FRootMetaType.BaseType = TypeOf(TObject) then
391:         break;
392:       FRootMetaType := FRootMetaType.BaseType;
393:     end;
394:
395:     // look for a couple of fields that we will need
396:     FRootHandleField := FRootMetaType.GetField('FInstanceTypeHandl
e',
BindingFlags.NonPublic or BindingFlags.Instance);
397:     if FRootHandleField = nil then
398:       raise EProxyError.Create(SCouldNotFindTypeHandle);
399:
400:     FRootParentField := FRootMetaType.GetField('FClassParent',
BindingFlags.NonPublic or BindingFlags.Instance);
401:     if FRootParentField = nil then
402:       raise EProxyError.Create(SCouldNotFindParent);
403:   end;
404:
405:   // add a metatype for this type we are working on and add a fiel
d
to the type
406:   LTypeBuilder := ATypeBuilder.DefineNestedType('@Meta' +
ATypeBuilder.Name,
407:     TypeAttributes.NestedPublic or TypeAttributes.BeforeFieldInit,

LBaseType);
408:
409:   // add attribute or two
410:   CreateBoolAttribute(LTypeBuilder,
TypeOf(System.CLSCompliantAttribute));
411:   CreateBoolAttribute(LTypeBuilder,
TypeOf(System.Runtime.InteropServices.ComVisibleAttribute));
412:
413:   // create our own instance field
414:   LInstanceField := LTypeBuilder.DefineField('@Instance',
LTypeBuilder,
415:     FieldAttributes.Public or FieldAttributes.Static);
416:
417:   // build constructor
418:   LConstructorBuilder :=
LTypeBuilder.DefineConstructor(MethodAttributes.Public or
MethodAttributes.HideBySig,
419:     CallingConventions.Standard, []);
420:   LILGenerator := LConstructorBuilder.GetILGenerator;
421:   with LILGenerator, OpCodes do
422:   begin
423:     // CODE TO BE GENERATED
424:     // inherited Create;
425:     // FInstanceTypeHandle := Self.TypeHandle;
426:     // FClassParent := {ParentClass}.@Instance; // only codegen if
parentclass has one
427:
428:     LBaseConstructor := LBaseType.GetConstructor([]);  // find the
base's create
429:     if LBaseConstructor = nil then
430:       raise EProxyError.Create(SCouldNotFindConstructor);
431:     Emit(Ldarg_0);                                          // pus
h
the instance
432:     Emit(Call, LBaseConstructor);        // emit a call to the pare
nt
constructor
433:
434:     Emit(Ldarg_0);                                          // pus
h
```

```
the instance
 435:      Emit(Ldtoken, AType);                          // push the hand
le
of the type
 436:      Emit(Stfld, FRootHandleFld);       // store the handle in th
e
root's field
 437:
 438:      // see if the base metatype has an instance field yet
 439:      LBaseInstanceField := LBaseType.GetField('@Instance',
BindingFlags.Public or BindingFlags.Static);
 440:      if LBaseInstanceField <> nil then
 441:      begin
 442:        Emit(Ldarg_0);                                        // pus
h
the instance
 443:        Emit(Ldsfld, LBaseInstanceField);                 // get t
he
parent info
 444:        Emit(Stfld, FRootParentField);              // store it i
nto
root field
 445:      end;
 446:
 447:      Emit(Ret);

    // fini
 448:   end;
 449:
 450:   // now create the class constructor
 451:   LTypeConstructorBuilder := LTypeBuilder.DefineTypeInitializer;
 452:   LILGenerator := LTypeConstructorBuilder.GetILGenerator;
 453:   with LILGenerator, OpCodes do
 454:   begin
 455:      // CODE TO BE GENERATED
 456:      // @Instance := @Meta{Class}.Create;
 457:
 458:      Emit(Newobj, LConstructorBuilder);     // create an instance of

the metaclass
 459:      Emit(Stsfld, LInstanceField);                 // store it in our
instance field
 460:
 461:      Emit(Ret);

    // fini
 462:   end;
 463:
 464:   // before we leave we had better actually create the type hadn't
we
 465:   Result := LTypeBuilder.CreateType;
 466: end;
 467:
 468: class procedure TProxyType.CodeGenConstructors(ABaseType:
System.Type; ATypeBuilder: TypeBuilder);
 469: var
 470:   LConstructors: array of ConstructorInfo;
 471:   LParameters: array of ParameterInfo;
 472:   LParamTypes: array of System.Type;
 473:   LConstructorBaseType: System.Type;
 474:   LConstructorBuilder: ConstructorBuilder;
 475:   LILGenerator: ILGenerator;
 476:   LConstructorNdx, LParameterNdx: Integer;
 477: begin
 478:   LConstructorBaseType := ABaseType;
 479:   while LConstructorBaseType <> nil do
 480:   begin
 481:
```

```
482:        // see if it has any constructors
483:        LConstructors := LConstructorBaseType.GetConstructors;
484:        if Length(LConstructors) <> 0 then
485:        begin
486:          for LConstructorNdx := Low(LConstructors) to
High(LConstructors) do
487:          begin
488:            with LConstructors[LConstructorNdx] do
489:            begin
490:              // copy the param and in turn their types
491:              LParameters := GetParameters;
492:              SetLength(LParamTypes, Length(LParameters));
493:              for LParameterNdx := Low(LParameters) to High(LParameter
s)
do
494:                LParamTypes[LParameterNdx] :=
LParameters[LParameterNdx].ParameterType;
495:
496:              // construct a constructor builder
497:              LConstructorBuilder :=
ATypeBuilder.DefineConstructor(Attributes,
498:                CallingConvention, LParamTypes);
499:            end;
500:
501:            // lets write some code
502:            LILGenerator := LConstructorBuilder.GetILGenerator;
503:            with LILGenerator, OpCodes do
504:            begin
505:              // CODE TO BE GENERATED
506:              // inherited Create({arg count depends on parentclass})
507:
508:              Emit(Ldarg_0);                                        //

push instance
509:              for LParameterNdx := 1 to Length(LParameters) do
510:                Emit(Ldarg_S, LParameterNdx);
//
push params
511:              Emit(Call, LConstructors[LConstructorNdx]);      // cal
l
the base ctr
512:
513:              Emit(Ret);

   // fini
514:            end;
515:          end;
516:
517:          // done
518:          break;
519:        end;
520:
521:        // move up a level
522:        LConstructorBaseType := LConstructorBaseType.BaseType;
523:      end;
524: end;
525:
526: class procedure TProxyType.CodeGenNotification(ABaseType:
System.Type; ATypeBuilder: TypeBuilder);
527: var
528:    LParamTypes: array of System.Type;
529:    LBaseNotificationMethod: MethodInfo;
530:    LMethodBuilder: MethodBuilder;
531:    LILGenerator: ILGenerator;
532:    LLabel: System.Reflection.Emit.Label;
533: begin
534:    // get the param list ready
535:    SetLength(LParamTypes, 2);
```

```
536:    LParamTypes[0] := TypeOf(Classes.TComponent);
537:    LParamTypes[1] := TypeOf(Classes.TOperation);
538:
539:    // see if we can find a notification method to call
540:    LBaseNotificationMethod := ABaseType.GetMethod('Notification',
541:        BindingFlags.Public or BindingFlags.NonPublic or
BindingFlags.Instance or
542:        BindingFlags.InvokeMethod, nil, LParamTypes, nil);
543:    if LBaseNotificationMethod <> nil then
544:    begin
545:
546:        // create a builder
547:        with LBaseNotificationMethod do
548:          LMethodBuilder := ATypeBuilder.DefineMethod(Name,
549:            MethodAttributes.FamORAssem or MethodAttributes.Virtual,
550:            CallingConvention, ReturnType, LParamTypes);
551:
552:        // let's write some code!
553:        LILGenerator := LMethodBuilder.GetILGenerator;
554:        with LILGenerator, OpCodes do
555:        begin
556:          // CODE TO BE GENERATED
557:          // Borland.Vcl.Design.Proxies.HandleNotification(Self,
AComponent, AOperation);
558:          // if Borland.Vcl.Classes.SendNotification(Self, AComponent,

AOperation) then
559:          //    inherited Notification(AComponent, AOperation);
560:
561:          Emit(Ldarg_0);                                              //

push instance
562:          Emit(Ldarg_1);                                   // push
component reference
563:          Emit(Ldarg_2);                                   // push what is
happening to it
564:          Emit(Call, FProxyNotificationMethod);     // call the proxy'
s
notify-wedge
565:
566:          Emit(Ldarg_0);                                              //

push instance
567:          Emit(Ldarg_1);                                   // push
component reference
568:          Emit(Ldarg_2);                                   // push what is
happening to it
569:          Emit(Call, FSendNotificationMethod);      // call classes'
sendnotification
570:
571:          LLabel := DefineLabel;
572:          Emit(Brfalse_S, LLabel);                       // if result is

false then...
573:
574:          Emit(Ldarg_0);                                              //

push instance
575:          Emit(Ldarg_1);                                   // push
component reference
576:          Emit(Ldarg_2);                                   // push what is
happening to it
577:          Emit(Call, LBaseNotificationMethod);          // call the

base's method
578:
579:          MarkLabel(LLabel);                                         //
...jump to here
```

```
580:
581:         Emit(Ret);

    // fini
582:     end;
583:   end;
584: end;
585:
586: class function TProxyType.FindRealType(var AType: System.Type):
Boolean;
587: begin
588:     // just in case were given a proxy type lets find the real type
589:     if AType is TProxyType then
590:       AType := AType.UnderlyingSystemType;
591:
592:     // see if we can find it in our list
593:     Result := FProxies.Contains(AType);
594: end;
595:
596: class function TProxyType.CreateSubType(ABaseType: System.Type;
597:     const AClassName: string; const AUnitName: string = ''):
System.Type;
598: var
599:     LTypeBuilder: TypeBuilder;
600:     LMetaType: System.Type;
601:     LMetaConstructor: ConstructorInfo;
602:     LProxyType: TProxyType;
603:     LNewType: System.Type;
604: begin
605:     // find the real type...  if we have been handed a proxytype,
instead of
606:     //   a 'realtype', then FindRealType will modify ABaseType so th
at
it
607:     //   points to the proxy's UnderlyingSystemType.
608:     FindRealType(ABaseType);
609:
610:     // create a type builder         ...remember each type must have
 a
unique name
611:     LTypeBuilder := FModuleBuilder.DefineType(Format(STestTypeName,
[FProxyTypeIndex]), TypeAttributes.Public, ABaseType);
612:     Inc(FProxyTypeIndex);
613:
614:     // find the first ancestor class that has constructors and copy
them
615:     CodeGenConstructors(ABaseType, LTypeBuilder);
616:
617:     // TODO: If the type is a TComponent desendent then we need to h
ook
notification
618:     CodeGenNotification(ABaseType, LTypeBuilder);
619:
620:     // quick make the type before it slips away again :-)
621:     LNewType := LTypeBuilder.CreateType;
622:     LProxyType := TProxyType.Create(LNewType, AClassName, AUnitName)
;
623:
624:     // make up a metaclass for the Delphi System unit
625:     LMetaType := CreateMetaSubType(ABaseType, LNewType, LTypeBuilder
);
626:     LMetaConstructor := LMetaType.GetConstructor([]);
627:     if LMetaConstructor = nil then
628:       raise EProxyError.Create(SCouldNotFindMetaConstructor);
629:
630:     // plug ourselves into the class delegator system so that our pr
oxy
type will
```

```
631:    //  be found when someone does a ClassInfo on this type/metatype
632:    SetClassDelegator(LProxyType, LMetaConstructor.Invoke([]));
633:
634:    // add it to the list of known 'live' proxies
635:    FProxies.Add(LNewType, LMetaType);
636:
637:    // return the proxy type
638:    Result := LProxyType;
639: end;
640:
641: class procedure TProxyType.SaveIt;
642: begin
643:    // caution: this is a one shot thing! once you call this you can
't
644:    //   create anymore proxy classes.
645:    FAssemblyBuilder.Save(STestFileName);
646: end;
647:
648: class procedure TProxyType.ChangeToProxyType(AType: System.Type);
649: begin
650:    // if it is already a proxy then complain...  if we have been
handed a
651:    //   proxytype, instead of a 'realtype', then FindRealType will
modify
652:    //   AType so that it points to the proxy's UnderlyingSystemType
.
653:    if FindRealType(AType) then
654:      raise EProxyError.Create(SAlreadyProxy);
655:
656:    // add the delegator
657:    SetClassDelegator(TProxyType.Create(AType, AType.Name,
AType.NameSpace));
658:
659:    // add it the proxy list
660:    FProxies.Add(AType, TypeOf(TClass(AType)));
661: end;
662:
663: class procedure TProxyType.DestroySubType(AType: System.Type);
664: begin
665:    // is it really subtyped?  if so then complain loudly...  if we
have been
666:    //   handed a proxytype, instead of a 'realtype', then FindRealT
ype
will
667:    //   modify AType so that it points to the proxy's
UnderlyingSystemType.
668:    if not FindRealType(AType) then
669:      raise EProxyError.Create(STypeNotSubType);
670:
671:    // remove it from the proxy list
672:    FProxies.Remove(AType);
673:
674:    // remove the delegator
675:    RemoveClassDelegator(AType);
676: end;
677:
678: class procedure TProxyType.RenameSubType(AType: System.Type;
679:    const AClassName: string; const AUnitName: string = '');
680: begin
681:    // is it really subtyped? (we call IsSubType because we don't wa
nt
the realtype)
682:    if not IsSubTyped(AType) then
683:      raise EProxyError.Create(STypeNotSubType);
684:
685:    // change the name
686:    TProxyType(AType).FClassName := AClassName;
687:    if AUnitName <> '' then
```

```
688:         TProxyType(AType).FUnitName := AUnitName;
689: end;
690:
691: class function TProxyType.FindProxy(AInstance: TObject): TProxyTyp
e;
692: var
693:    LType: System.Type;
694: begin
695:    // find the type
696:    LType := AInstance.ClassInfo;
697:
698:    // make sure it is what we need otherwise complain
699:    if not (LType is TProxyType) then
700:      raise EProxyError.Create(STypeNotSubType);
701:    Result := TProxyType(LType);
702: end;
703:
704: function TProxyType.CreateMethod(const AMethodName: string):
TMethodCode;
705: var
706:    LMethodCode: TMethodCode;
707: begin
708:    LMethodCode := TMethodProxy(FMethods[AMethodName]);
709:    if LMethodCode = nil then
710:    begin
711:      LMethodCode := TMethodProxy.Create(Self, AMethodName);
712:      FMethods.Add(AMethodName, LMethodCode);
713:    end;
714:    Result := LMethodCode;
715: end;
716:
717: procedure TProxyType.RenameMethod(const AMethodCode: TMethodCode;
const AMethodName: string);
718: begin
719:    // make sure it is a method proxy
720:    if not (AMethodCode is TMethodProxy) then
721:      raise EProxyError.Create(SMethodNotMethodProxy);
722:
723:    // remove, rename and re-add
724:    FMethods.Remove(AMethodCode.Name);
725:    TMethodProxy(AMethodCode).Rename(AMethodName);
726:    FMethods.Add(AMethodName, AMethodCode);
727: end;
728:
729: procedure TProxyType.DestroyMethod(const AMethodCode: TMethodCode)
;
730: begin
731:    // make sure it is a method proxy
732:    if not (AMethodCode is TMethodProxy) then
733:      raise EProxyError.Create(SMethodNotMethodProxy);
734:
735:    // remove and clear
736:    FMethods.Remove(AMethodCode.Name);
737:    TMethodProxy(AMethodCode).Clear;
738: end;
739:
740: class function TProxyType.GetMethodAddress(AClass: TClass; const
AName: string; out ACode: TMethodCode): Boolean;
741: var
742:    LType: System.Type;
743: begin
744:    // assume failure
745:    ACode := nil;
746:
747:    // find the class' type
748:    LType := AClass.ClassInfo;
749:    Result := LType is TProxyType;
750:
```

```
751:     // keep looking but only if the type is a TProxyType
752:     while LType is TProxyType do
753:     begin
754:
755:        // see if there is a method
756:        ACode := TMethodCode(TProxyType(LType).FMethods.Item[AName]);
757:        if ACode <> nil then
758:           break;
759:
760:        // still nothing?  then look at the parent class
761:        AClass := AClass.ClassParent;
762:        LType := AClass.ClassInfo;
763:     end;
764: end;
765:
766: class function TProxyType.GetMethodProp(AInstance: TObject;
APropInfo: TPropInfo; out AMethod: TMethod): Boolean;
767: var
768:    LInstanceRef: TInstanceRef;
769:    LMethodRef: TObject;
770: begin
771:    // find the instance
772:    LInstanceRef := TInstanceRef(FInstances.Item[AInstance]);
773:    Result := LInstanceRef <> nil;
774:
775:    // do our thing?
776:    if Result then
777:    begin
778:
779:       // find the property
780:       LMethodRef := LInstanceRef.Props.Item[APropInfo];
781:
782:       // if nothing
783:       if LMethodRef = nil then
784:          AMethod := TMethod.Empty
785:       else
786:          AMethod := TMethod(LMethodRef);
787:
788:       // I guess it worked
789:       Result := True;
790:    end;
791: end;
792:
793: class function TProxyType.SetMethodProp(AInstance: TObject;
APropInfo: TPropInfo; const AMethod: TMethod): Boolean;
794: var
795:    LInstanceRef: TInstanceRef;
796: begin
797:    // something we care about?
798:    Result := (AMethod.Data = nil) or IsProxyClass(AMethod.Data);
799:    if Result then
800:    begin
801:
802:       // find the instance
803:       LInstanceRef := TInstanceRef(FInstances.Item[AInstance]);
804:       if LInstanceRef = nil then
805:       begin
806:          LInstanceRef := TInstanceRef.Create;
807:          FInstances.Add(AInstance, LInstanceRef);
808:       end;
809:
810:       // adding?
811:       if not AMethod.IsEmpty then
812:          LInstanceRef.Props[APropInfo] := AMethod.Clone
813:
814:       // removing?
815:       else
816:       begin
```

```
817:        // poof!
818:        LInstanceRef.Props.Remove(APropInfo);
819:
820:        // if there are no props defined then get rid of the instanc
e
itself
821:        if LInstanceRef.Props.Count = 0 then
822:          FInstances.Remove(AInstance);
823:      end;
824:    end;
825: end;
826:
827: class function TProxyType.GetUnitName(ATypeInfo: TTypeInfo; out
AUnitName: string): Boolean;
828: begin
829:    // assume success
830:    Result := True;
831:
832:    // go find the right type and get its proxy, if there is one
833:    AUnitName := TClass(ATypeInfo).ClassInfo.NameSpace;
834: end;
835:
836: class procedure TProxyType.HandleNotification(Sender: TObject;
AComponent: TComponent; Operation: TOperation);
837: begin
838:    // remove it from our list
839:    if Operation = opRemove then
840:      TProxyType.FInstances.Remove(AComponent);
841: end;
842:
843: { TMethodProxy }
844:
845: constructor TMethodProxy.Create(AProxyType: TProxyType; const ANam
e:
string);
846: begin
847:    inherited Create;
848:    FProxyType := AProxyType;
849:    FName := AName;
850: end;
851:
852: procedure TMethodProxy.Clear;
853: begin
854:    FProxyType := nil;
855:    FName := '';
856: end;
857:
858: procedure TMethodProxy.Rename(Value: string);
859: begin
860:    FName := Value;
861: end;
862:
863: function TMethodProxy.get_ProxyType: TProxyType;
864: begin
865:    Result := FProxyType;
866: end;
867:
868: function TMethodProxy.GetCustomAttributes(AInherit: Boolean):
TObjects;
869: begin
870:    Result := GetCustomAttributes(nil, AInherit);
871: end;
872:
873: function TMethodProxy.GetCustomAttributes(AttributeType: System.Ty
pe;
Inherit: Boolean): TObjects;
874: begin
875:    SetLength(Result, 0);
```

```
876: end;
877:
878: function TMethodProxy.IsDefined(AttributeType: System.Type; Inheri
t:
Boolean): Boolean;
879: begin
880:    Result := False;
881: end;
882:
883: function TMethodProxy.get_DeclaringType: System.Type;
884: begin
885:    Result := FProxyType;
886: end;
887:
888: function TMethodProxy.get_MemberType: MemberTypes;
889: begin
890:    Result := MemberTypes.Method;
891: end;
892:
893: function TMethodProxy.get_Name: string;
894: begin
895:    Result := FName;
896: end;
897:
898: function TMethodProxy.get_ReflectedType: System.Type;
899: begin
900:    Result := nil;
901: end;
902:
903: { Unit functions }
904:
905: function CreateSubClass(AAncestor: TClass; const AClassName: strin
g;
906:    const AUnitName: string): TClass;
907: begin
908:    Result := TClass(TProxyType.CreateSubType(AAncestor.ClassInfo,
AClassName, AUnitName));
909: end;
910:
911: resourcestring
912:    SNoValidConstructor = 'No valid constructor found for %s.';
913:
914: function ConstructSubClass(AClass: TClass; AParams: array of
TObject): TObject;
915: var
916:    LParameterNdx: Integer;
917:    LParamTypes: array of System.Type;
918:    LConstructor: ConstructorInfo;
919: begin
920:    SetLength(LParamTypes, Length(AParams));
921:    for LParameterNdx := Low(AParams) to High(AParams) do
922:       if AParams[LParameterNdx] = nil then
923:          LParamTypes[LParameterNdx] := TypeOf(TObject)
924:       else
925:          LParamTypes[LParameterNdx] := AParams[LParameterNdx].ClassIn
fo;
926:    LConstructor := AClass.ClassInfo.GetConstructor(LParamTypes);
927:    if LConstructor = nil then
928:       raise EProxyError.CreateFmt(SNoValidConstructor,
[AClass.ClassName]);
929:    Result := LConstructor.Invoke(AParams)
930: end;
931:
932: function ConstructComponent(AClass: TComponentClass; AOwner:
TComponent = nil): TComponent;
933: var
934:    LParamTypes: array of System.Type;
935:    LConstructor: ConstructorInfo;
```

```
936: begin
937: //Result := AClass.Create(AOwner); // Corbin note: we need this to
work...soon.....
938: //Exit;
939:    SetLength(LParamTypes, 1);
940:    LParamTypes[0] := TypeInfo(TComponent);
941:    LConstructor := AClass.ClassInfo.GetConstructor(LParamTypes);
942:    if LConstructor = nil then
943:    begin
944:      { Try a parameterless constructor }
945:      SetLength(LParamTypes, 0);
946:      LConstructor := AClass.ClassInfo.GetConstructor(LParamTypes);
947:      if LConstructor <> nil then
948:      begin
949:        Result := TComponent(LConstructor.Invoke([]));
950:        if AOwner <> nil then
951:          AOwner.InsertComponent(Result);
952:      end
953:      else
954:        raise EProxyError.CreateFmt(SNoValidConstructor,
[AClass.ClassName]);
955:    end
956:    else
957:      Result := TComponent(LConstructor.Invoke([AOwner]))
958: end;
959:
960: procedure DestroySubClass(AInstance: TObject);
961: begin
962:    DestroySubClass(AInstance.ClassType);
963: end;
964:
965: procedure DestroySubClass(AClass: TClass);
966: begin
967:    TProxyType.DestroySubType(AClass.ClassInfo);
968: end;
969:
970: procedure RenameSubClass(AInstance: TObject; const AClassName,
AUnitName: string);
971: begin
972:    RenameSubClass(AInstance.ClassType, AClassName, AUnitName);
973: end;
974:
975: procedure RenameSubClass(AClass: TClass; const AClassName, AUnitNa
me:
string);
976: begin
977:    TProxyType.RenameSubType(AClass.ClassInfo, AClassName, AUnitName
);
978: end;
979:
980: function IsProxyClass(AInstance: TObject): Boolean;
981: begin
982:    Result := IsProxyClass(AInstance.ClassType);
983: end;
984:
985: function IsProxyClass(AClass: TClass): Boolean;
986: begin
987:    Result := TProxyType.IsSubTyped(AClass.ClassInfo);
988: end;
989:
990: procedure ChangeToProxyClass(AInstance: TObject);
991: begin
992:    ChangeToProxyClass(AInstance.ClassType);
993: end;
994:
995: procedure ChangeToProxyClass(AClass: TClass);
996: begin
```

```
 997:     TProxyType.ChangeToProxyType(AClass.ClassInfo);
 998: end;
 999:
1000: function CreateSubClassMethod(AInstance: TObject; const AMethodNam
e:
string): TMethodCode;
1001: begin
1002:    Result := TProxyType.FindProxy(AInstance).CreateMethod(AMethodNa
me);
1003: end;
1004:
1005: procedure RenameSubClassMethod(AInstance: TObject; const AMethodCo
de:
TMethodCode; const AMethodName: string);
1006: begin
1007:    TProxyType.FindProxy(AInstance).RenameMethod(AMethodCode,
AMethodName);
1008: end;
1009:
1010: procedure DestroySubClassMethod(AInstance: TObject; const
AMethodCode: TMethodCode);
1011: begin
1012:    TProxyType.FindProxy(AInstance).DestroyMethod(AMethodCode);
1013: end;
1014:
1015: procedure HandleNotification(Sender: TObject; AComponent: TCompone
nt;
Operation: TOperation);
1016: begin
1017:    TProxyType.HandleNotification(Sender, AComponent, Operation);
1018: end;
1019:
1020: procedure SaveIt;
1021: begin
1022:    TProxyType.SaveIt;
1023: end;
1024:
1025: end.
```